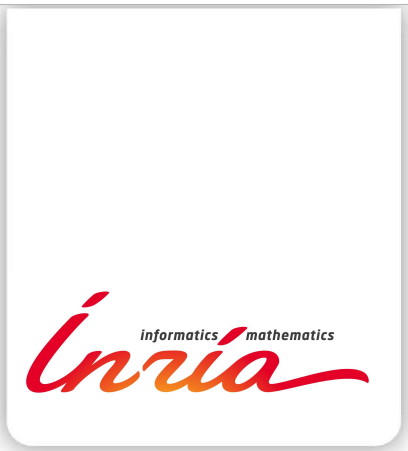
Towards an automatic generation of dense linear algebra solvers on parallel architectures

Marc Baboulin, Joel Falcou, Ian Masliah

RESEARCH REPORT
N° 8615
Octobre 2014
Project-Team Postale

ISSN 0249-6399 ISRN INRIA/RR--8615--FR+ENG



Towards an automatic generation of dense linear algebra solvers on parallel architectures

Marc Baboulin*, Joel Falcou†, Ian Masliah‡

Project-Team Postale

Research Report n° 8615 — Octobre 2014 — 17 pages

Abstract: The increasing complexity of new parallel architectures has widened the gap between adaptability and efficiency of the codes. As high performance numerical libraries tend to focus more on performance, we wish to address this issue using a C++ library called NT^2. By analyzing the properties of the linear algebra domain that can be extracted from numerical libraries and combining them with architectural features, we developed a generic approach to solve dense linear systems on various architectures including CPU and GPU. We have then extended our work with an example of a least squares solver based on semi-normal equations in mixed precision that cannot be found in current libraries. For the automatically generated solvers, we report performance comparison with state-of-the-art codes, showing that it is possible to obtain a generic code with a high-level interface (similar to MATLAB) that can run either on CPU or GPU and that does not generate significant overhead.

Key-words: Numerical libraries, generative programming, active libraries, GPU computing, dense linear systems, mixed precision algorithms

* Inria and Université Paris-Sud, France (marc.baboulin@inria.fr).
† Inria and Université Paris-Sud, France (joel.falcou@lri.fr).
‡ Inria and Université Paris-Sud, France (ian.masliah@lri.fr).

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Vers une génération automatique de solveurs d'algèbre linéaire dense sur les architectures parallèles

Résumé : La complexité croissante des nouvelles architectures parallèles a augmenté l'écart entre l'adaptabilité et l'efficacité des codes. Les bibliothèques numériques de calcul haute performance ayant tendance à se concentrer sur l'efficacité, nous proposons une solution à ce problème en utilisant une bibliothèque de calcul en C++ appelée NT^2. En analysant les propriétés du domaine d'algèbre linéaire qui peuvent être extraites des bibliothèques numériques et en les combinant avec des caractéristiques architecturales, nous avons développé une approche générique pour résoudre les systèmes linéaires sur diverses architectures comprenant des CPU et des GPU. Ce travail a été étendu avec l'ajout d'un solveur de moindres-carrés basé sur les équations semi-normales en précision mixte, cet algorithme n'étant pas disponible dans les bibliothèques actuelles. Pour les solveurs générés automatiquement, nous avons pu observer des performances équivalentes aux codes couramment utilisés, ce qui a permis de montrer qu'il est possible d'obtenir un code générique avec une interface haut-niveau (similaire à MATLAB) qui peut tourner sur CPU ou GPU sans surcoût de temps.

Mots-clés : calcul numérique, programmation générative, bibliothèques actives, calcul sur GPU, systèmes linéaires denses, algorithmes en précision mixte.

Introduction

A major concern when developing dense linear algebra software is to propose a user-friendly Application Programming Interface (API) that can retain performance of BLAS-like [23] optimized routines. Moreover, with the increase in parallelism and heterogeneity as well as the ever increasing data-communication costs, numerical libraries often require to be modified or redesigned in order to take advantage of new features in parallel architectures [35]. In our study we consider more specifically the dense linear algebra libraries LAPACK [2] (serial library for CPU processors) and MAGMA [44] (for Graphics Processing Units). The disparity between these libraries that are targeting different architectures illustrate one of the issues in designing optimized linear algebra software. While being able to maintain a similar interface for the routines, the code and structure of all algorithms ported from LAPACK to MAGMA had to be rewritten to match the architectural features and the programming language of the accelerator. Furthermore, these libraries are implemented using low-level languages like C or FORTRAN and thus cannot provide a high-level interface that would be closer to the specification language of the numerical linear algebra practitioner without losing in performance. This issue is represented by the *abstraction/-efficiency trade-off* problem where raising the abstraction level with object-oriented and generic programming techniques is obtained at the cost of performance, which inhibits the flexibility and adaptability of libraries.

Some solutions have been proposed in recent years but they tend to solve partially the *abstraction/efficiency trade-off* problem. The method followed by the Formal Linear Algebra Methods Environment (FLAME) with the Libflame library [46] is a good example. It offers a framework to develop dense linear solvers through the use of algorithmic skeletons [15] and an API which is more user-friendly than LAPACK while giving satisfactory performance results. Another approach is the one followed in recent years by C++ libraries built around *expression templates* [48] or other *generative programming* [20] principles for high-performance computing. Examples of such libraries are Armadillo [16] and MTL [27]. Armadillo provides good performance with BLAS and LAPACK bindings and an API close to MATLAB [36] for simplicity. However it does not provide a generic solver like the MATLAB routine *linsolve* that can analyze the matrix type and choose the correct routine to call from the LAPACK library. It also does not support GPU computations which are becoming mandatory for medium to large dense linear algebra problems. In a similar way, while MTL can topple the performance of vendor-tuned codes, it does not offer *linsolves*-like implementation or GPU support. Other examples of libraries with similar content include Eigen [30] , Flens [34], Ublas [49] and Blaze [32].

Our objective in this paper is to provide a solution to the problems of *portability* and *adaptability* on new computer architectures. It is designed on top of NT^2 [25, 26], a scientific library written in C++. NT^2 provides a MATLAB-inspired API and its implementation is based on a *meta-programming* technique known as "expression templates" [48]. The contributions of this paper are the following.

- We propose an architecture aware binding between NT^2 and LAPACK/MAGMA based on markers to dispatch between the different architectures and runtime back-ends in a extensible way.
- We provide an implementation of *linsolve* (in reference to the MATLAB routine) that takes into account both hardware and algorithmic features to select and generate the proper chain of calls of optimized LAPACK/MAGMA routines from the high-level C++ code, mapping over 160 kernels. Note that the support for different factorizations (*QR*, *Cholesky*, *LU*, *SVD*) is also provided in NT^2 to facilitate the development of new solvers.

- An example of automatically generated solver (for different architectures) that does not exist in current libraries is given. This is illustrated by a linear least squares solver in mixed-precision arithmetic that achieves performance similar to optimized routines. For this solver, we show that *linsolve* does not generate a size-able overhead compared to direct calls to LAPACK or MAGMA.

This paper is organized as follows: in Section 1, we describe various programming techniques that can be found to combine algorithmic and architectural features in libraries. The methods that we used in NT^2 are then introduced. They enable us to achieve re-use and adaptability of library codes while preserving performance. In Section 2.1, we demonstrate how to combine these techniques in developing efficient dense linear algebra software. Then, as an example of application, we present in Section 2.3 the code generation of a mixed-precision linear least squares solver for which we give performance comparisons on CPU and GPU using respectively the QR routines from LAPACK and MAGMA. To our knowledge such a solver does not exist in public domain libraries LAPACK, PLASMA [43] and MAGMA. Concluding remarks are given in Section 3.

1 Generative programming for designing numerical libraries

1.1 Optimization approaches based on a configuration space

As stated in Section , developing complex linear algebra software is a non trivial task due to the large amount of both algorithmic and architectural requirements. These combined factors create a *configuration space* containing the various configurations available for a given system. Deciding the correct combination of factors from a *configuration space* will then ensure optimal performance.

Compiler techniques based on *iterative compilation* [45], where several optimizations from a *configuration space* are tested and the best one is selected, is a classical technique to improve performance. Similarly, such methods also exist at the library level.

An example of these methods can be found in the ATLAS [50] library which is based on using optimized binaries. Each function's binary is generated during the installation phase with the *iterative compilation* technique. The generation process is accelerated by a hierarchical tuning system. In this system, the lower level functions are subject to a large selection process ensuring their optimal performance. High-level functions like BLAS 3 can then exploit feedback from the previous steps of the configuration process.

A second method is based on a performance analysis at runtime. For instance, a system like StarPU [3] uses a monitored runtime system in which the performance of each function on a given hardware configuration is monitored in real-time. This monitoring allows StarPU to select the most optimized version of the algorithm by changing its parameters (tiling size, number of iterations,...) or the targeted architecture (CPU, GPU, hybrid).

Both methods described above are valid approaches in the field of high performance computing. However, in our case, we aim at providing a library level system for such exploration [47] that will complement the compilers work. One way to do this is to use *generative programming.*

1.2 Generative programming in software development

Generative programming consists of bringing the benefits of automation to software development. Following this paradigm, a model can be drawn to implement the different components [18] of a system. It is then possible to build a generator that will combine these components based on a generative domain model. This generator (or *configuration knowledge*) will ensure the transition from a *configuration space* with domain-specific concepts [29] and features to a *solution space* that encapsulates expertise at the source-code level. The code generation process will be hidden from the end-user by various *meta-programming* techniques which turn the user interface into a simple and clean API where few to none details about the algorithms and structures are visible.

Template *meta-programming* is a classical *generative programming* technique in which templates are used by a compiler to generate temporary source code. It is then merged with the rest of the source code and then finally compiled. The output of these templates includes compile-time constants, data structures, and complete functions. The use of templates can be thought of as compile-time execution that enables us to implement domain-specific optimizations. The technique is used by a number of languages, the most well-known being C++ [1], D [14], Haskell [41] and OCaml [42].

1.3 Domain engineering methods for active libraries

We call *active libraries* [17] a technique which structures a set of *generative programming* and *meta-programming* methods to solve the *abstraction/efficiency trade-off problem* mentioned in Section . The main idea is to perform high-level optimization based on a semantic analysis of the code before any real compilation process. Such informations and transformations are then carried on by a meta-language that allows the developer to embed meta-informations in the source code itself, helping compilers to generate a better code by using these semantic informations. Active libraries are often implemented as *Domain specific embedded language* (or DSEL), which are languages implemented inside another host language. Designing DSEL is often easier as they reuse existing compilers and rely on domain dependent analysis to generate efficient code.

Czarnecki proposed a methodology called *Domain Engineering Method for Reusable Algorithmic Libraries* [19] (or DEMRAL) based on the techniques described previously. DEMRAL can be seen as a specialization of a paradigm like object-oriented programming, aspect programming or model driven engineering [40]. While we can find a large number of algorithms (N) and implementations for distinct architectures (P), the problem is that combining them can result in a large number of code to write ($N * P$). Using DEMRAL, only N generic algorithms and P data structure descriptions are needed since the generator will choose the correct domain-specific implementation from the *configuration space* with the help of the *configuration knowledge*.

The DEMRAL methodology provides a high re-usability, allowing components to be customized while retaining the efficiency of statically configured code [37]. We extend it by adding an architectural-aware design (or AA-DEMRAL, depicted in Figure 1) with a domain specific architecture description and a specialized generator. This enables us to properly dispatch the various parametric sub-components by first generating them with an architectural marker (see Architecture generative component in Figure 1)

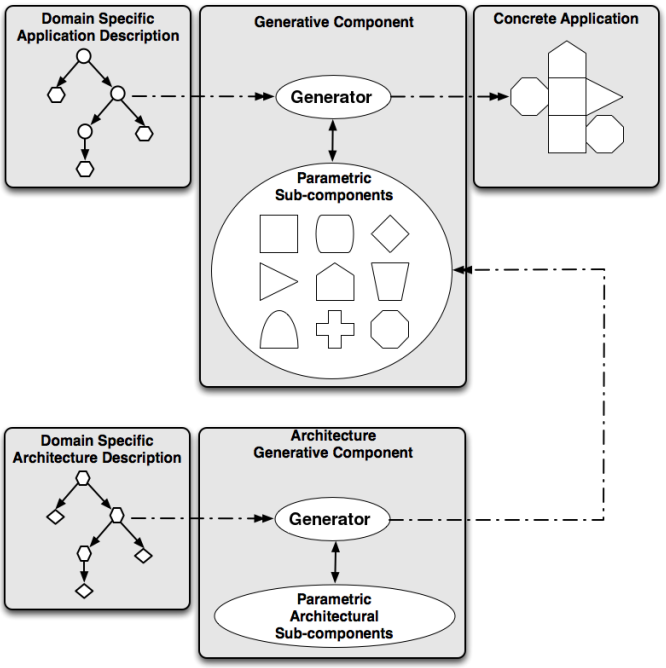


Figure 1: Overview of the AA-*DEMRAL* process

2 Application to linear algebra solvers

In this section, we describe our approach to automatically generate linear algebra solvers on parallel architectures. Our answer stems from the programming techniques combined with a proper *configuration space* and *smart containers* for data management on GPU.

2.1 Linear system solvers

The first step to build *linsolve* is to identify the key properties of the *configuration space* and how to represent them. Once this analysis is done, we can refine these properties into high-level abstractions that will parameterize *linsolve*. These abstractions will then be used to define the *configuration knowledge* necessary to ensure the transition to the *solution space*. Concerning dense linear systems, we can identify 3 main properties that need to be taken into account: matrix structure, condition number, and targeted architecture.

A matrix structure can be divided into subcategories that can be identified statically (data type, storage scheme, matrix type and storage format). The data type and storage scheme parameters are already identified through the problem domain, respectively being scalar entries (real, double or single/double complex) and a dense matrix. The storage format is defined by NT^2 and shares a common interface with FORTRAN77 (column-major arrays). The matrix types correspond to the different ones available in the numerical libraries LAPACK and MAGMA (*e.g., general, symmetric, hermitian...*).

From the analysis of these subcategories, we consider the data and matrix types as important parameters. As they impact the underlying function call that needs to be generated, they will

both represent a level of the *configuration space*.

The second domain corresponds to the conditioning of the system. In current numerical libraries, the linear solvers are usually based on LU or QR factorizations in fixed precision, or mixed-precision algorithms [6] with iterative refinement. It is not possible to identify statically if a system is ill-conditioned since it requires expensive computations which are not manageable at compile-time. Furthermore, it would be too costly to estimate the condition number at run-time for mixed-precision routines since it requires the factored form of the matrix (the LAPACK function `gecon` estimates the reciprocal condition number but requires the LU form bringing the cost to $\theta(n^3)$ for an $n * n$ matrix). However, since current dense linear algebra libraries propose mixed precision routines, it needs to be part of the *configuration space*.

The last key domain of our solver is the dispatch between different architectures. As explained in Section , the architectural features of a GPU result in a very different language compared to a CPU. The solution we used to solve this abstraction problem is to provide through the use of a DSEL (Section 1.2) a common syntax between CPU and GPU routines. Using architecture aware binding, we can then freely choose between NT^2 and LAPACK/MAGMA to dispatch on the different back-ends in an extensible way. It is now possible to define a grammar that encapsulates these ideas into a *configuration space* (see Section 1.1).

Table 1: Configuration space parameters levels

0-Matrix type	general | band | diagonal | symmetric | positive definite
1-Data type	float | double | single/double complex
2-Precision	fixed | mixed-precision
3-Conditioning	no information | ill-conditioned
4-Storage scheme	general | packed
5-Architecture	CPU | GPU

Most of the parameters we can access are defined by the user and therefore configurable at the API level. In MATLAB, the *linsolve* routine does not take into account the data type, and the matrix type needs to be defined in a parameter structure containing the different matrix properties recognized (lower/upper triangular, upper Hessenberg, symmetric, positive definite, rectangular). While creating a matrix in NT^2, the user has the possibility to define the matrix and data type which are optimized as meta-data properties of the matrix, using the following instruction:

```
nt2::table<double,nt2::symmetric_> a;
```

When calling *linsolve*, he will then have the possibility to give additional information on the conditioning of the matrix either as a parameter of the system:

```
x = nt2::linsolve(a,b,nt2::ill_conditioned_);
```

or of the matrix:

```
x = nt2::linsolve(nt2::ill_conditioned_(a),b);
```

It is also possible to ask for complementary information as output like the reciprocal condition number returned by LAPACK :

```
nt2::tie(x,r) = nt2::linsolve(a,b);
```

Once this is done, *linsolve* will be able to parse the *configuration space* by reading out the nested domain-specific features while assigning default values to the unspecified ones.

2.2 Memory management for hybrid computation

When using GPU-based systems, we need ensuring data consistency between the different physical memories. In this section, we discuss how these techniques are used jointly with *linsolve*. We can discern the two most common approaches to CPU and GPU containers. The first one is to statically define the locality of the container which is done in the Thrust library [11], while the second uses a dynamic approach like in SkePU [24].

The memory management mechanism in a dynamic approach allows to change the locality of a container and reallocate the data. The container then needs to manage the memory and ensure consistency between data and locality. In this situation, it is not possible to statically define the locality of a container and therefore doing a dispatch using the locality in our generation phase. Therefore, our approach consists of adding an architectural tag similarly to the matrix type tag on our container (default locality is CPU). The purpose of this method is to enable the user to write programs using GPU to GPU functions in a transparent way.

```
nt2::table<double,nt2::gpu_> a;
```

It is then possible to ensure the transitions from CPU to GPU memory by using explicitly the tag. This does not prevent the decision-making process of the solver when no locality tag is given by the user. The solver can generate a GPU code performing data transfers from CPU to GPU and the reverse. The generation process will choose the architecture based on a combination of factor being mainly the matrix size and the algorithm. The GPU tag can also hold complementary informations passed as template parameters of the tag.

The definition of container locality being static, it is easier to define a data efficient memory management unit. Let's use the following scenario as an example :

```
x = nt2::linsolve(a,b);
```

From here, we can apply different strategies depending on the locality of x, a and b. In a situation where all three containers are on GPU/CPU memory there will be no locality problem as various data are located on the same device. However, the scenario where x is on the GPU (respectively CPU) while a and b are on the CPU (resp. GPU) will generate a conflict. The rules to solve locality conflicts are static and do not depend on a runtime. Therefore, the priority will be given to the locality of the result x to ensure consistency between the data locality and container locality.

Experiments were carried out on a system using 2 sockets of Intel Xeon E5645 2.40GHz and a Tesla C2075. We consider single precision random square matrices of size 2000, 10000 and 20000 and we solve a system of linear equations $Ax = b$ using the LU factorization. The light grey bars in Figure 2 correspond to the following call made in NT^2 through *linsolve* that can run either on CPU or GPU.

```
x = nt2::linsolve(a,b);
```

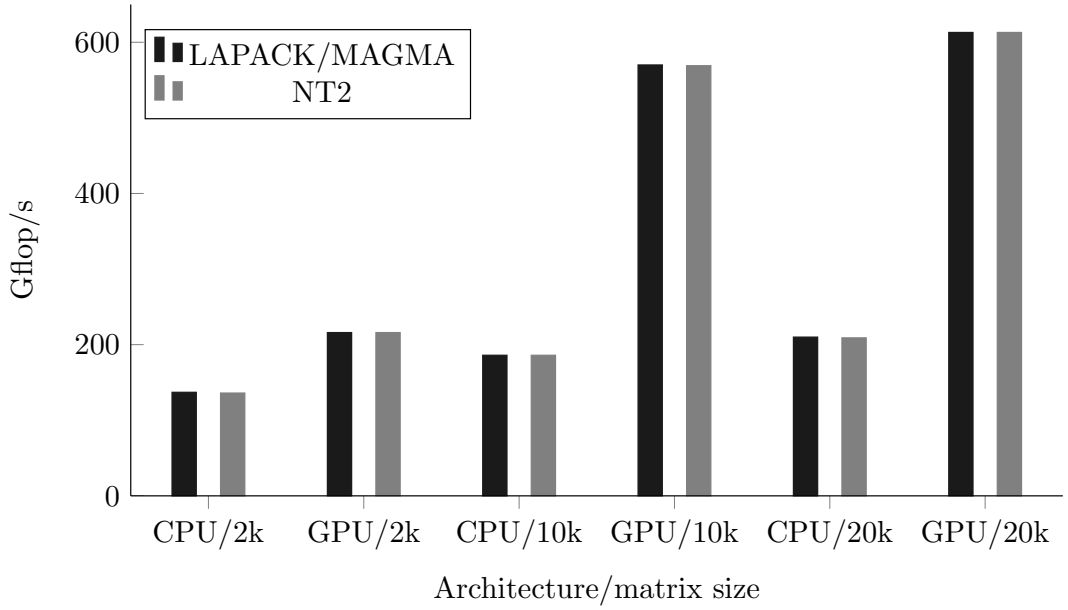


Figure 2: Performance comparaison between LAPACK/MAGMA routines and generated codes via NT2 for general dense linear system solution

The dark grey bars correspond to C++ calls to either the LAPACK function `sgesv` or the MAGMA function `magma_sgesv`. These results show that automatically generated routines do not exhibit any overhead compared to direct calls to LAPACK or MAGMA. Note that the performance of all others generated routines also does not incur any overhead.

2.3 Application to linear least squares

In this section we illustrate how the *generative programming* method described in Section 1 can be used to generate automatically new implementations of algorithms that do not exist in current libraries and achieve satisfactory performance compared with existing solvers.

2.3.1 Solving least squares by semi-normal equations

We consider the overdetermined full rank linear least squares (LLS) problem $\min_{x\in\mathbb{R}^n} \|Ax-b\|_2$, with $A \in \mathbb{R}^{m\times n}, m \geq n$ and $b \in \mathbb{R}^m$.

The most classical methods for solving linear least squares problems are based on the QR factorization or the normal equations. The latter method is twice cheaper (mn^2 vs $2mn^2$ operations) but the condition number is then proportional to $\text{cond}(A)^2$ [13, p. 49]). However if A can be saved, we can also use the semi-normal equations (SNE) method where we solve the system

$$R^T Rx = A^T b,$$

where R is the triangular factor from the QR factorization of A (this is a straightforward reformulation of the normal equations). It is shown in [12] that, similarly to the normal equations method, the forward error bound involves a factor $cond(A)^2$, even if we use a R-factor that is of better quality than the Cholesky factor because it has been computed via a backward stable algorithm. However, as explained in [13, p. 126 and p. 250], the accuracy of the SNE method

can be improved by using the corrected semi-normal equations method (CSNE) that consists in adding one step of fixed precision iterative refinement to the SNE as follows:

1. Let $\tilde{x}$ solve $R^T Rx = A^T b$
2. Compute $\tilde{r} = b - A\tilde{x}$
3. Solve $R^T Rw = A^T \tilde{r}$
4. Corrected solution $y = \tilde{x} + w$

It is shown in [31, p. 392] that, if $\text{cond}(A)^2 u \leq 1$ (u being the unit roundoff), then the forward error bound for the CSNE method is similar to that of a backward stable method (and even smaller when $r = Ax - b$ is small). In that case, the CSNE method is more satisfactory than the SNE method but this is not true for all A. In the following we propose to use the CSNE method to solve LLS in mixed precision.

2.3.2 Mixed-Precision Corrected Semi-Normal Equation

The efficiency of mixed precision algorithms has been proved on linear systems based on the LU factorization with results that can reach up to 90% [6] of floating point computational rate in the lowest precision on current architectures. The method to solve mixed precision CSNE (or MCSNE) consists of first solving the factorization in single precision (ε_s) (if the matrix is not too ill-conditioned) with the computational cost of $\theta(mn^2)$ and then refine the solution in double precision (ε_d) where operations cost $\theta(n^2)$. Iterative refinement [22] is a method that produces a correction to the computed solution by iterating on it. Each k^{th} iteration in this process consists of computing the residual $r_k = b - Ax_{k-1}$, solving the new system $Ad_k = r_k$, and adding the correction $x_{k+1} = x_k + d_k$. Mixed precision iterative refinement will work as long as the condition number of the least squares problem [8] is smaller than the inverse of the lower precision used (*i.e.* here 10^8).

Algorithm 2.1 Mixed precision CSNE

Compute $A = QR$	(ε_s)
Solve $R^T x = A^T b$	(ε_s)
Solve $Rx_0 = x$	(ε_s)
`do k = 1,2,...`	
$r_k = b - Ax_{k-1}$	(ε_d)
Solve $R^T x = r_k$	(ε_s)
Solve $Rd_k = x$	(ε_s)
$x_k = x_{k-1} + d_k$	(ε_d)
`check convergence`	

The purpose of having an automatic generation is to be able to keep the expressiveness of the algorithmic version of a code while achieving high performance that can topple state of the art libraries. The MCSNE routine has been generated using the following NT2 instructions.

The first step of Algorithm 2.1 in NT^2 is implemented in line 8 of Listing 1, while the second and third steps are performed by two calls to *linsolve* in lines 11 and 12. Note that the code is similar in terms of syntax and number of instructions to what would be written in MATLAB.

Listing 1: NT2 implementation for MCSNE

```
table<double> mcsne(table<double> const& A, table<double> const& B)
{
  double anrm = lange(A,'I');
  double cte  = anrm*Eps<double>()*nt2::sqrt(width(a));

  table<float> SA = cast<float>(A);

  table<float,upper_triangular_> SR = triu( qr(SA,no_pivot_) );
  table<float> SX = mtimes(trans(SA),cast<float>(B)));

  SX = linsolve(trans(SR),SX);
  SX = linsolve(SR       ,SX);

  table<double> X = cast<double>(SX);
  table<double> E = B - mtimes(A,X);

  std::size_t i = 0;

  do
  {
    SX = cast<float>(mtimes(trans(A),cast<float>(E)));
    SX = linsolve(trans(SR),SX);
    SX = linsolve(SR       ,SX);

    E = cast<double>(SX)

    double RNRM = maximum(abs(E(_)));

    X += E;
    double XNRM = maximum(abs(X(_)));

    E = B - mtimes(A,X);
    i++;
  } while( !(RNRM < XNRM*cte) && (i<max_iter));

  return X;
}
```

Once the solver for MCSNE has been coded using NT^2, it becomes possible to add it to *linsolve* as a dispatch case of mixed precision solver for overdetermined linear systems. This would result in the following call :

```
x = nt2::linsolve(a,b,nt2::mixed_precision_);
```

2.3.3 Performance results for MCSNE

Benchmarks were carried out using 2 sockets of Intel Xeon E5645 2.40GHz (peak Gflop/s is 230) and a Tesla C2075 (peak Gflop/s is 1030.4). We used Intel MKL [33] version 10.2.3, MAGMA 1.3 with CUDA 5.0 [38] and gcc 4.8 [28]. The random test problems were generated using the method described in [39]. Performance results include data transfers between CPU and GPU and data are stored on the GPU memory.

In Figure 3 we compare the performance on the CPU for MCSNE with the LAPACK routine `xgels` that solves the LLS problem with a QR factorization without column pivoting. The results show that performance of MCSNE is only 10% less that the rate of `sgels`. Note that this represents around 75% of the peak performance of a matrix-matrix multiply in single precision (routine `sgemm`). We use random matrices and the iterative refinement converged in less than 4 iterations.

On the GPU, the performance of MCSNE in NT^2 is depicted in Figure 4. It also approaches 90% of the performance of **magma_sgels** on GPU while being near twice faster than the routine

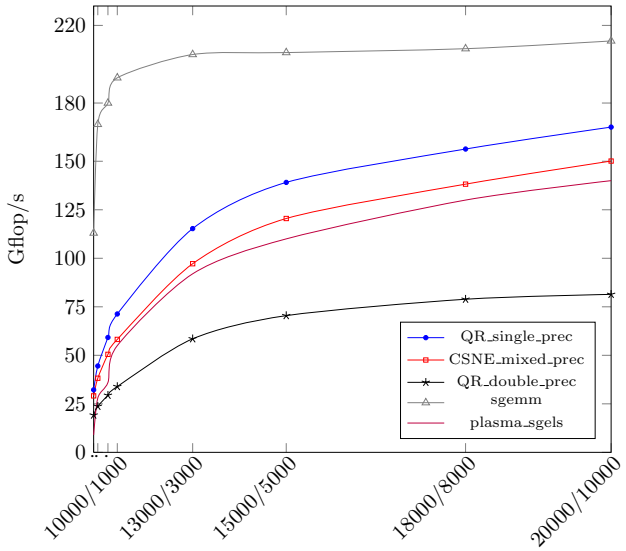


Figure 3: Performance results of Generated code on CPU of gels (QR solver) and mcsne

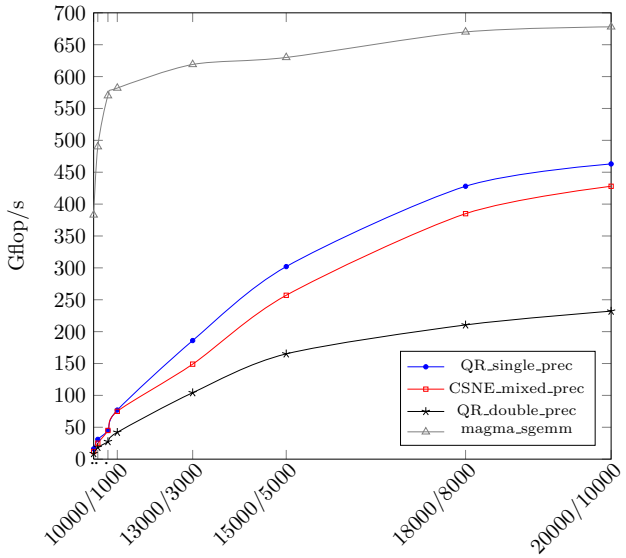


Figure 4: Performance results of Generated code on GPU of gels (QR solver) and mcsne

in double precision. The behavior of MCSNE when compared with QR solvers in double and single precision is similar to what was observed in [4, p. 15] for the LU factorization.

3 Conclusion

Combining the large number of algorithms available in numerical libraries and architectural requirements into a generic solver for dense linear systems is a complex task. We showed that *generative programming* is a valid software development approach for addressing these issues while maintaining a high level of performance. Our contribution furthers the work in *active libraries* by providing a viable way to render our software architecture-aware. Performance results illustrate that for both existing routines like those in *linsolve* and new ones such as MCSNE, the delivered performance is close to what state of the art libraries achieve.

The other interesting result is that software like NT^2 can quickly prototype new algorithms while providing support for various architectures. With NT^2, we reach a good combination of high-level codes for linear algebra problems that gives good speedups and offers the users enough expressiveness to describe the problem in the most efficient way.

Future work includes support for more architectures like Intel Xeon Phi, with work on new algorithms that provide good performances while not being available in numerical libraries like randomized algorithms [5, 9] or communication-avoiding algorithms [7] for dense linear systems. Moving to sparse problems is also a possibility where libraries like Cups [10] or VexCL [21] provide an interesting approach. Raising the level of expressiveness stays a major concern while trying to add content in NT^2.

References

[1] D. Abrahams and A. Gurtovoy. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Addison-Wesley Professional, 2004.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 3 edition, 1999.

[3] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] M. Baboulin. Fast and reliable solutions for numerical linear algebra solvers in high-performance computing. `http://tel.archives-ouvertes.fr/tel-00967523`, 2012. Habilitation thesis - University of Paris-Sud.

[5] M. Baboulin, D. Becker, and J. Dongarra. A Parallel Tiled Solver for Dense Symmetric Indefinite Systems on Multicore Architectures. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS 2012)*, pages 14–24, 2012.

[6] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.

[7] M. Baboulin, S. Donfack, J. Dongarra, L. Grigori, A. Rémy, and S. Tomov. A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. In *International Conference on Computational Science (ICCS 2012)*, volume 9 of *Procedia Computer Science*, pages 17–26. Elsevier, 2012.

[8] M. Baboulin, J. Dongarra, S. Gratton, and J. Langou. Computing the conditioning of the components of a linear least-squares solution. *Numerical Linear Algebra with Applications*, 16(7):517–533, 2009.

[9] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov. Accelerating linear system solutions using randomization techniques. *ACM Trans. Math. Softw.*, 39(2), 2013.

[10] N. Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations. `http://cusp-library.googlecode.com`, 2012. Version 0.3.0.

[11] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems*, 7, 2011.

[12] A. Björck. Stability analysis of the method of semi-normal equations for least squares problems. *Linear Algebra and its Applications*, 88/89:31–48, 1987.

[13] A. Björck. *Numerical methods for least squares problems*. Siam, 1996.

[14] W. Bright. D language Templates revisited. `http://dlang.org/templates-revisited.html`.

[15] M. I. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[16] S. Conrad. Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, Australia, October 2010.

[17] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries. In *Generic Programming*, pages 25–39. Springer, 2000.

[18] K. Czarnecki and U. W. Eisenecker. Components and generative programming. In *Software Engineering—ESEC/FSE'99*, pages 2–19. Springer, 1999.

[19] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[20] K. Czarnecki, K. Østerbye, and M. Völter. Generative programming. In *Object-Oriented Technology ECOOP 2002 Workshop Reader*, pages 15–29. Springer, 2002.

[21] D. Demidov. VexCL: Vector expression template library for OpenCL. `http://github.com/ddemidov/vexcl`, 2012.

[22] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy. Error bounds from extra-precise iterative refinement. *ACM Trans. Math. Softw.*, 32(2):325–351, 2006.

[23] J. Dongarra. Basic Linear Algebra Subprograms Technical Forum Standard. *Int. J. of High Performance Computing Applications*, 16(1), 2002.

[24] J. Enmyren and C. W. Kessler. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM, 2010.

[25] P. Esterie, J. Falcou, M. Gaunard, J. T. Lapresté, and L. Lacassagne. The numerical template toolbox: A modern C++ design for scientific computing. *Journal of Parallel and Distributed Computing*, 2014.

[26] J. Falcou, J. Sérot, L. Pech, and J. T. Lapresté. Meta-programming applied to automatic SMP parallelization of linear algebra code. In *Euro-Par 2008–Parallel Processing*, pages 729–738. Springer, 2008.

[27] P. Gottschling, D. S. Wise, and M. D. Adams. Representation-transparent matrix algorithms with scalable performance. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 116–125, New York, NY, USA, 2007. ACM Press.

[28] B. J. Gough and R. M. Stallman. *An Introduction to GCC*. Network Theory Ltd., 2004.

[29] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *ACM SIGPLAN Notices*, volume 41, pages 291–310. ACM, 2006.

[30] G. Guennebaud, B. Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[31] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 2 edition, 2002.

[32] K. Iglberger, G. Hager, J. Treibig, and U. Rüde. Expression templates revisited: a performance analysis of current methodologies. *SIAM Journal on Scientific Computing*, 34(2):C42–C69, 2012.

[33] Intel. Math Kernel Library (MKL). `http://www.intel.com/software/products/mkl/`.

[34] M. Lehn. FLENS. `http://http://www.mathematik.uni-ulm.de/~lehn/FLENS/`, 2013.

[35] P. Luszczek, J. Kurzak, and J. Dongarra. Looking back at dense linear algebra software. *Journal of Parallel and Distributed Computing*, 2013.

[36] MATLAB. *version 8.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2013.

[37] D. R. Musser, G. J. Derge, and A. Saini. *STL tutorial and reference guide: C++ programming with the standard template library*. Addison-Wesley Professional, 2009.

[38] NVIDIA. *NVIDIA CUDA C Programming Guide*, 04/16/2012. Version 4.2.

[39] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, 8(1):43–71, 1982.

[40] D. C Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):0025–31, 2006.

[41] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

[42] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[43] University Tennessee. PLASMA users' guide, parallel linear algebra software for multicore architectures, version 2.3. `http://icl.cs.utk.edu/plasma/software/`, 2010.

[44] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5&6):232–240, 2010.

[45] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 204–215. IEEE, 2003.

[46] F. G. Van Zee, E. Chan, R. A. Van de Geijn, E. S. Quintana-Orti, and G. Quintana-Orti. The libflame library for dense matrix computations. *Computing in science & engineering*, 11(6):56–63, 2009.

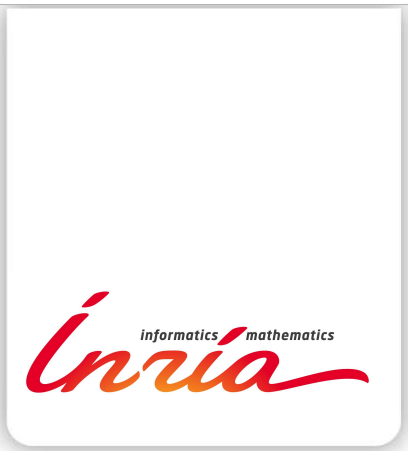
[47] T. L. Veldhuizen and E. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[48] Todd Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.

[49] J. Walter and M. Koch. The boost uBLAS library. `http://www.boost.org/libs/numeric/ublas`, 2002.

[50] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

Contents



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399